C

CWS/6/6 CORR.
原 文：英文
日 期：2018 年 9 月 14 日

世界知识产权组织标准委员会（CWS）

第六届会议
2018 年 10 月 15 日至 19 日，日内瓦

关于网络应用程序接口的新产权组织标准

国际局编拟的文件

导 言

1. 产权组织标准委员会（CWS）在于 2017 年 5 月 29 日至 6 月 2 日举行的第五届会议上，讨论了依据 XML4IP 工作队的讨论结果为知识产权信息和文献 Web 服务编写建议的必要性。若干代表团分享了它们在 Web 服务方面的经验和计划（见文件 CWS/5/22 第 89 段至第 90 段）。

2. 标准委员会在会议上同意创建第 56 号任务，并将这项新任务分派给 XML4IP 工作队，这项任务的说明转录如下：

> “为支持机器对机器通讯的数据交换编写建议，重点是
>
> (i) 采用 JavaScript 对象表示法（JSON）和/或 XML 的消息格式、数据结构和数据字典；以及
>
> (ii) 资源的统一资源标识符（URI）命名约定。”

（见文件 CWS/5/22 第 91 段至第 93 段。）

3. 产权组织国际局于 2018 年 5 月举办了知识产权行政管理用信通技术策略和人工智能问题知识产权局会议。会议讨论依据文件 WIPO/IP/ITAI/GE/18/3 进行，该文件载有 40 项建议，见 http://www.wipo.int/meetings/en/details.jsp?meeting_id=46586。有关应用程序接口（API）的两项建议建议 38 和建议 39 转录如下，以供参阅。

建议 38：探讨更加先进的方法，实现与国际系统的整合及系统集中化。打造集中化服务作为示范或样板项目，采用开放式标准 API，用于分类和标准数据的传播及知识产权局与区域/国际知识产权体系之间交易数据的交流。

建议 39：共享在线服务信息（申请、后续交易等），旨在识别可通过 API 提供的常见交易和服务项目，实现系统之间的互操作性，包括第三方解决方案提供商开发的系统。

4. 各代表团在上述会议上表示，很多知识产权局已在使用应用程序接口，并计划把它们更多的服务通过应用程序接口提供。代表团还承认应用程序接口在各知识产权局之间的一致性对于数据交换的效率至关重要，特别是对于第三方专利管理系统提供商来说，它们从商业角度出发可能不会愿意支持各局采用不同标准。向各代表团通报了 XML4IP 工作队正在编写网络应用程序接口的新建议，各代表团同意积极参与标准委员会工作队的工作，为新的应用程序接口产权组织标准编写最终提案，供委员会本届会议审议通过。（见文件 WIPO/IP/ITAI/GE/18/5。）

新标准工作草案

5. 为了执行第 56 号任务，工作队通过其电子论坛维基进行了四轮讨论，举行了若干次在线会议以及 2018 年 5 月在莫斯科的一次实体会议。工作队还对工作草案进行了数次更新，最新的工作草案 0.7 版转录于本文件附件以供参阅，该版仅以英文提供。

标准的目标

6. 工作队商定标准旨在提供有关应用程序接口的建议，以便为在网络中统一处理和交换知识产权数据提供便利。

7. 工作队还商定标准的目的应是为了：

- 通过建立统一的 Web 服务设计原则确保一致性；
- 提升 Web 服务伙伴之间的数据互用性；
- 通过统一的设计鼓励可重用性；
- 通过相关 XML 资源中明确界定的命名空间政策加强业务部门之间的数据命名灵活性；
- 加强安全的信息交流；
- 提供可供其他组织使用的适当内部业务流程作为附加值服务；
- 整合内部业务流程，并把它们与业务伙伴进行动态链接。

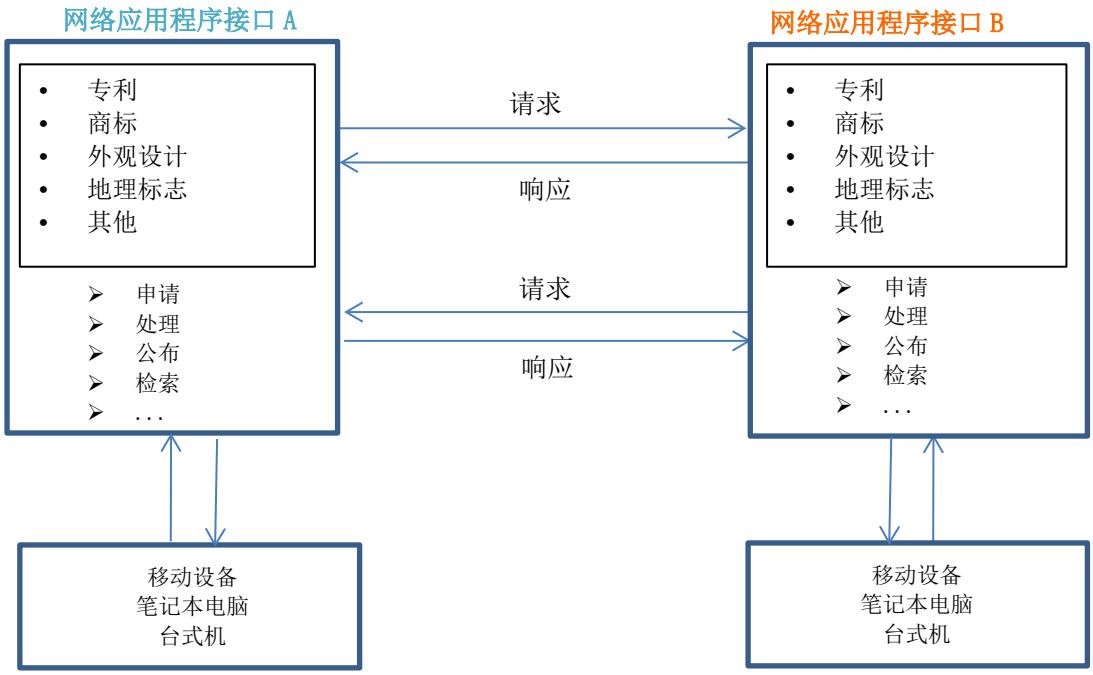
标准的范围

8. 工作队认为，标准应向需要通过网络应用程序接口管理、存储、处理、交换和传播知识产权数据的知识产权局和其他组织提供指引。使用该项标准可以统一的方式使网络应用程序接口开发得到简化和加快，使网络应用程序接口之间的互用性得到提升。

9. 标准对两种类型的 Web 服务提供了建议：

- “RESTful 网络应用程序接口”：一组基于 REST 结构范式的 Web 服务，通常使用 JSON 或 XML 传送数据；以及
- “SOAP 网络应用程序接口”：一组基于 SOAP 的 Web 服务，必须使用 XML 作为负载格式。

10. 标准还将涵盖知识产权局与其申请人或数据用户之间以及各知识产权局之间通过设备到设备和设备到软件应用的连接所进行的通信。



标准的结构

11. 最新的 0.7 版工作草案由一个主体部分和七个附件组成。附件一至六还未完成，因为正在等待来自工作队成员的进一步意见。尤其是附件二和附件三将包含基于各知识产权局的实践或计划做出的建议，这些实践和计划是关于通过网络应用程序接口公开（将要公开）的知识产权数据资源。

12. 此外，附件四——RESTful 网络应用程序接口合同范本包含基于 RESTful 应用程序接口建模语言（RAML）的文档范本草案作为一份独立文档。国际局计划编写另一份基于开放应用程序接口规范（OAS）和 Web 服务描述语言（WSDL）的合同范本，知识产权局可对其进行微调，以实施它们自己的应用程序接口。

未决议题

13. 国际局在 2018 年 6 月 14 日举办了在线会议，来自 7 个知识产权局的与会专家讨论了以下未决议题：

- 资源名称的单复数问题，如 person、persons 还是 people；
- RESTful 网络应用程序接口的文档范本基于 RAML 还是 OAS；
- 负载格式为 XML 还是 JSON；
- 把 XML 数据转换为 JSON 格式和 JSON 模式；
- RESTful 网络应用程序接口安全模型；以及
- 知识产权数据资源列表。

14. 考虑到会议讨论和有关这些议题的进一步评论意见，工作草案建议：

- 采用复数形式，而不是单数形式，因为大多数知识产权局倾向于或使用复数形式，即使用 persons。
- RESTful 网络应用程序接口的文档范文基于 RAML 和 OAS。
- 负载格式为 XML 和 JSON。
- 在提供 JSON 模式前，利用 BadgerFish 把 XML 转换为 JSON。考虑到知识产权局逐渐开始更频繁地使用 JSON 格式，工作队认为应基于 WIPO 标准 ST.96 对 JSON 模式进行开发。但由于 JSON 模式没有议定的行业标准，因此工作队继续监测行业对 JSON 模式的开发情况。工作队商定命名约定采用小驼峰拼写法，如 applicantName，XML 组件根据 WIPO 标准 ST.96 采用大驼峰拼写法，如 ApplicantName。工作队还商定对把 ST.96 XSD 进一步转换为 JSON 模式进行讨论。
- 高级别 RESTful 网络应用程序接口安全模型依据国际局的提案建立，具体实施留由各知识产权局负责，因为它们应遵循其自己的安全指导。
- 知识产权数据资源名称和相关信息列表。为了制订该列表，邀请各知识产权局提供它们（计划）公开和希望使用其他知识产权局数据的应用程序接口资源列表。

15. 此外，工作队讨论了基于新标准的通用应用程序接口的好处和必要性。一个工作队成员知识产权局建议开发一个 RESTful 网络应用程序接口，以根据 WIPO 标准 ST.27 提供专利法律状态事件数据。建议在第六届会议上讨论是否需要开发通用网络应用程序接口以及涉及哪个业务领域，如专利法律状态数据交换，以及各知识产权局之间如何为开发开展合作。

进一步的讨论和开发

16. 确定了对以下各项进行进一步讨论和开发：

- 基于 WIPO 标准 ST.96 的 RESTful 网络应用程序接口 JSON 规范。
- 进一步与 OData 协调一致，Odata 是一项行业标准，已为越来越多的供应商采用，尽管它实施起来比较复杂。
- 需要进行一致性测试以确保标准的多种实施方式。甚至可以使用安装有软件的测试台，任何人都可对其进行调用，并对它的一致性进行量化，如 W3C 为判断 HTML 一致性而使用的 https://validator.w3.org/。WIPO 标准 ST.96 还提供了用于验证知识产权局实施模式与 ST.96 之间兼容性的工具。
- 更多使用 RAML/OAS 的 RESTful 网络应用程序接口合同范本，以及更多使用 WSDL 的 SOAP 网络应用程序接口合同范本，这些合同范本将基于标准中确定的规则，以便知识产权局直接照原样下载和使用，或是便于进行扩展。通过这项工作可使知识产权局的应用程序接口与标准一致，并将知识产权局的实施费用降至最低。
- 要制订完成资源和查询参数列表，说明各统一资源标识符所对应的查询参数、请求主体、HTTP 头部信息和 HTTP 动词，以便通过网络应用程序接口使用知识产权局提供的服务。
- 应商定数据格式和响应内容，如它是否包含结果数量、命名空间、复杂检索语法等。

– 进一步开发新功能，如更新自动通知功能。

17. 请标准委员会：

(a) 注意本文件及其附件的内容；

(b) 对转录于本文件附件的工作草案发表评论意见；

(c) 对上文第 15 段所述的开发通用应用程序接口进行讨论；并

(d) 要求 XML4IP 工作队提交有关网络应用程序接口新标准的提案供委员会在第七届会议审议。

[后接附件]

WIPO STANDARD ST.XX

RECOMMENDATIONS FOR WEB API ON INTELLECTUAL PROPERTY DATA

Working Draft - version 0.7

Editorial Note prepared by the International Bureau

This Working Draft is prepared by the XML4IP Task Force and shared for information at the sixth session of the CWS only in English. This Draft will be further updated in due course and the final draft will be submitted for consideration by the CWS at its seventh session.

TABLE OF CONTENTS

1. INTRODUCTION

1. This Standard provides recommendations on Application Programming Interface (API) to facilitate the processing and exchange of Intellectual Property (IP) data in harmonized way over the Web.

2. This Standard is intended to:

- ensure consistency by establishing uniform web service design principles;
- improve data interoperability among web service partners;
- encourage reusability through unified design;
- promote data naming flexibility across business units through a clearly defined namespace policy in associated XML resources;
- promote secure information exchange;
- offer appropriate internal business processes as value-added services that can be used by other organizations;
- integrate its internal business processes and dynamically link them with business partners.

2. DEFINITIONS AND TERMINOLOGY

3. For the purpose of this Standard, the expression:

- The term “HyperText Transfer Protocol (HTTP)” is intended to refer to the application protocol for distributed, collaborative, and hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web. HTTP functions as a request–response protocol in the service oriented computing model.
- The term “Application Programming Interfaces” (API) means software components that provide a reusable interface between different applications that can easily interact to exchange data.
- The term “Representational State Transfer (REST)” describes a set of architectural principles by which data can be transmitted over a standardized interface, i.e. HTTP. REST does not contain an additional messaging layer and focuses on design rules for creating stateless services.
- The term “Simple Object Access Protocol (SOAP)” means a protocol for sending and receiving messages between applications without confronting interoperability issues. SOAP defines a standard communication protocol (set of rules) specification for XML-based message exchange. SOAP uses different transport protocols, such as HTTP and SMTP. The standard protocol HTTP makes it easier for SOAP model to tunnel across firewalls and proxies without any modifications to the SOAP protocol.
- The term “Web Service” means a method of communication between two applications or electronic machines over the World Wide Web (WWW) and Web Services are of two kinds: REST and SOAP.
- “RESTful Web API” means a set of Web Services based on REST architectural paradigm and typically use JSON or XML to transmit data.
- “SOAP Web API” means a set of SOAP Web Services based on SOAP and mandate the use of XML as the payload format.
- The term “Web Services Description Language (WSDL)" means a W3C Standard that is used with the SOAP protocol to provide a description of a Web Service. This includes the methods a Web Service uses, the parameters it takes and the means of locating Web Services etc.
- The term “Service Contract” (or Web Service Contract) means a document that expresses how the service exposes its capabilities as functions and resources offered as a published API by the service to other software programs; the term “REST API documentation” is interchangeably used for the Service Contract for RESTful Web APIs.
- The term “Service Provider” means a Web Service software exposing a Web Service.
- The term “Service Consumer” means the runtime role assumed by a software program when it accesses and invokes a service. More specifically, when the program sends a message to a service capability expressed in the service contract. Upon receiving the request, the service begins processing and it may or may not return a corresponding response message to the service consumer.
- The term “camelcase” is either the lowerCamelCase or the UpperCamelCase naming convention.
- The term Kebab-case is one of the naming conventions where all are lowercase with hyphens “-“ separating words, for example a-b-c.
- The term “Open Standards” means the standards that are made available to the general public and are developed (or approved) and maintained via a collaborative and consensus driven process. “Open Standards” facilitate interoperability and data exchange among different products of services and are intended for widespread adoption.
- Uniform Resource Identifier (URI) identifies a resource and Uniform Resource Locator (URL) is a subset of the URIs that include a network location.
- The term “Entity Tag (ETag)” means an opaque identifier assigned by a web server to a specific version of a resource found at a URL. If the resource representation at that URL ever changes, a new and different ETag is assigned. ETags can be compared quickly to determine whether two representations of a resource are the same.
- The term “Service Registry” means a network-based directory that contains available services.

- The term “Semantic Versioning” means a versioning scheme where a version is identified by the version number MAJOR.MINOR.PATCH, where:

    - MAJOR version when you make incompatible API changes,
    - MINOR version when you add functionality in a backwards-compatible manner and
    - PATCH version when you make backwards-compatible bug fixes.

3. NOTATIONS

3.1. General notations

4. The following notations are used throughout this document:

- <>: Indicates a placeholder descriptive term that, in implementation, will be replaced by a specific instance value.
- “ ”: Indicates that the text included in quotes must be used verbatim in implementation.
- { }: Indicates that the items are optional in implementation.
- `Courier font`: Indicates keywords and source code.

3.2. Rule identifiers

5. All design rules are normative. Design rules are identified through a prefix of [XX-nn].

- The value “XX” is a prefix to categorize the type of rule as follows:
- WS for SOAP Web API design rules
- RS for RESTful Web API design rules
- CS for both SOAP and RESTful WEB API design rule
- The value “nn” indicates the next available number in the sequence of a specific rule type. The number does not reflect the position of the rule, in particular, for a new rule. A new rule will be placed in the relevant context. For example, the rule identifier [WS-4] identifies the fourth SOAP Web API design rule. The rule [WS-4] can be placed between rules [WS-10] and [WS-11] instead of following [WS-3] if that is the most appropriate location for this rule.
- The rule identifier of the deleted rule will be kept while the rule text will be replaced with “Deleted”.

4. SCOPE

6. This Standard aims to guide the Intellectual Property Offices (IPOs) and other Organizations that need to manage, store, process, exchange and disseminate IP data using Web APIs. It is intended that by using this Standard, the development of Web APIs can be simplified and accelerated in a harmonized manner and interoperability among Web APIs can be enhanced.

7. This Standard intends to cover the communications between IPOs and their applicants or data users, and between IPOs through connections between devices-to-devices and devices-to-software applications.

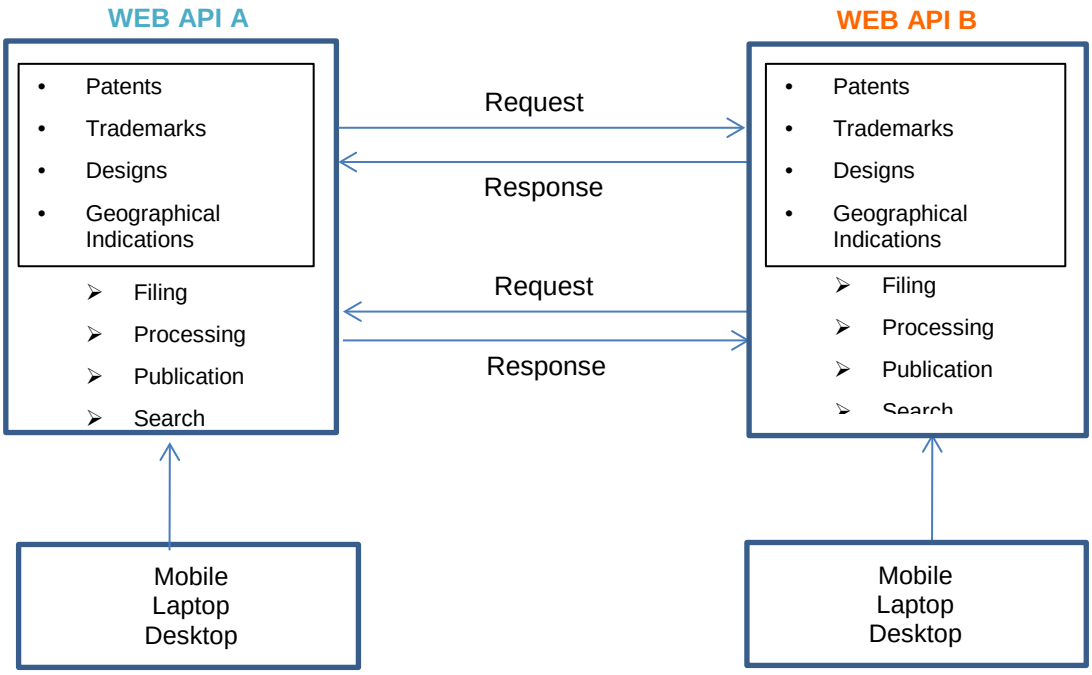


Fig. 1 Scope of the Standard

8. This Standard is to provide a set of design rules and conventions for RESTful Web APIs and SOAP Web APIs; list of IP data resources which will be exchanged or exposed; model API documentation or service contract, which can be used for customization, describing message format, data structure and data dictionary in JSON and/or XML based on WIPO Standard ST.96 and mock-up (reference) APIs to be used by IPOs.

9. This Standard provides model Service Contracts for SOAP Web APIs using WSDL and, for RESTful Web APIs using the REST API Modeling Language (RAML) and Open API Specification (OAS). A Service Contract also defines or refers to data types for interfaces (see the Section "Data Type Convention" below). This Standard recommends three types of interfaces: REST-XML (XSD), REST-JSON and SOAP-XML (XSD).

10. This Standard excludes the following:

(a) Binding to specific implementation technology stacks and commercial off-the-shelf (COTS) products;

(b) Binding to specific architectural designs (for example, Service Oriented Architecture (SOA) or Microservice Architecture);

(c) Binding to specific algorithms such as algorithms for the calculation of ETag, i.e. calculation of a unique identifier for a specific version of a resource (for example, used for caching).

5. WEB API DESIGN PRINCIPLES

11. Both RESTful Web APIs and SOAP Web APIs have proven their ability to meet the demands of big organizations as well as to service the small-embedded applications in production. When choosing between RESTful and SOAP, the following aspects can be considered:

- Security, e.g., SOAP has WS-Security while REST does not have any security specification;
- ACID Transaction, .e.g., SOAP has WS-AT specification while REST does not have a relevant specification ;
- Architectural style, e.g., Microservices and Serverless Architecture Style use REST while SOA uses SOAP web services;
- Flexibility;
- Bandwidth constraints;
- Guaranteed delivery, e.g. SOAP offers WS-RM while REST does not have a relevant specification.

12. The following design principles should be respected when a Web API is designed:

Service Contract Standardization – Standardizing the service contracts is the most important design principle because the contracts allow governance and a consistent service design. A service contract should be easy to implement and understand. A service contract consists of metadata that describes how the service provider and consumer will interact. Metadata also describes the conditions under which those parties are entitled to engage in an interaction. It is recommended that service contracts include:

- Functional requirements: what functionality the Service provides and what data it will return, or typically a combination of the two;
- Non-functional requirements: information about the responsibility of the providers for providing their functionality and/or data, as well as the expected responsibilities of the consumers of that information and what they will need to provide in return. For example, a consumer's availability, security, and other quality of service considerations.

13. A service contract should take into consideration the following design principles:

Service Loose Coupling – Clients and services should evolve independently. Applying this design principle requires:

- Service versioning – Consumers bound to a Web API version should not take the risk of unexpected disruptions due to incompatible API changes.
- The service contract should be independent of the technology details.

Service Abstraction– The service implementation details should be hidden. The API Design should be independent of the strategies supported by a server. For example, for the REST Web Service, the API resource model should be decoupled from the entity model in the persistence layer.

Service Statelessness – Services should be scalable.

Service Reusability – A well-designed API should provide reusable services with generic contracts. In this regard, this Standard provides a model service contract.

Service Autonomy – The Service functional boundaries should be well defined.

Limit server usage – The server resource usage should be limited in multi-tenant systems.

Using Standards as a Foundation – The API Should follow industry standards (such as IETF, ISO, and OASIS) wherever applicable, naturally favoring them over locally optimized solutions.

Pick-and-choose Principle – It is not required to implement all the API design rules. The design rules should be chosen based on the implementation of each concrete case.

14. In addition, the following principles should be respected especially with regard to the RESTful Web APIs

(a) Cacheable - Responses explicitly indicate their cacheability;

(b) Resource identification in requests - Individual resources are identified in requests; for example using URIs in Web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client.

(c) Hypermedia as the engine of application state (HATEOAS) - Having accessed an initial URI for the REST application—analogous to a human Web user accessing the home page of a website—a REST client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs.

(d) Resource manipulation through representations - When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.

(e) Self-descriptive messages - Each message includes enough information to describe how to process the message.

(f) Design with the objective that the API will eventually be accessible from the public internet, even if there are no plans to do so at the moment.

(g) Use a common authentication and authorization pattern, preferably based on existing security components: avoid creating a bespoke solution for each API.

(h) Least Privilege - Access and authorization should be assigned to API consumers based on the minimal amount of access they need to carry out the functions required.

(i) Maximize entropy (randomness) of security credentials by using API Keys rather than username and passwords for API authorization, as API Keys provide an attack surface that is more challenging for potential attackers.

(j) Balance performance with security with reference to key life times and encryption / decryption overheads.

6. RESTFUL WEB API

15. A RESTful Web API allows requesting systems to access and manipulate textual representations of Web resources using a uniform and predefined set of stateless operations.

6.1. URI Components

16. RESTful Web API s use URIs to address resources. According to RFC 3986, an URI syntax should be defined as follows:

URI = <scheme> "://" <authority> "/" <path> {"?" query}

For example, https://wipo.int/api/v1/patent?orderBy=id&offset=10

```
  _____/________/__________/____________________/
    |      |          |          |
scheme authority  path  query parameters
```

17. The forward slash “/” character is used in the path of the URI to indicate a hierarchical relationship between resources.

[RS-01] The forward slash “/” MUST NOT be included in URLs as it does not add any semantic values, but may cause confusion.

18. URIs are case sensitive except for the scheme and host parts. For example, although `https://wipo.int/api/my-resource/uniqueId` and `https://wipo.INT/my-resource/uniqueId` are the same, `https://wipo.int/my-resource/uniqueid` is not. For the resource names, the kebab-case and the lowerCamelCase conventions provide good readability and maps the resource names to the entities in the programming languages with simple transformation. For the query parameters, the lowerCamelCase should be used. For example, `https://wipo.int/api/v1/inventors?firstName=John`. Resource names, query parameters, matrix parameters are all case sensitive.

19. A RESTful Web API may have arguments :

in the query parameter; for example, /inventors?id=1;

in the matrix parameter; for example, /inventors;id=1;

in the URI path segment parameter, for example, /inventors/1; and

In the request payload such as part of a json body.

Except of the aforementioned argument types, which are part of the URI, an argument can also be part of the request payload.

[RS-02] Resources name MUST be consistent in their naming pattern and the lowercase or kebab case naming convention SHOULD be used.

[RS-03] Query parameters MUST be consistent in their naming pattern naming and the lowerCamelCase convention SHOULD be used. Query parameter MAY be abbreviated.

20. A Web API endpoint must comply with `IETF RFC 3986` and should avoid potential collisions with page URLs for the website hosted on the root domain. A Web API needs to have one exact entry point to consolidate all requests. In general, there are two patterns of defining endpoints:

- As the first path segment of the URI, for example: https://wipo.int/api/v1/
- As subdomain, for example: https://api.wipo.int/v1/

[RS-04] The URL pattern for a Web API MUST contain the word “`api`” in the URI.

21. Matrix parameters are an indication that the API is complex with multiple levels of resources and sub-resources. This is against the design principles. Moreover, matrix parameters are not standard and they apply to a particular path element while query parameters apply to the request as a whole. An example of matrix parameters is the following: `https://api.wipo.int/v1/path;param1=value1;param2=value2.`

[RS-05] Matrix parameters MUST NOT be used.

6.2. Status Codes

22. A Web API must consistently use the meaning of HTTP status codes as described in IETF RFCs. The recommended HTTP Status Codes are listed in ANNEX VI – should be used.

[RS-06] A Web API MUST consistently use the meaning of HTTP status codes as described in IETF RFCs and the recommended codes in Annex VI SHOULD be used.

6.3. Pick-and-choose Principle

23. A Service Contract should be tolerant to unexpected parameters (in the request, query parameters or matrix parameters) but raise error in case of malformed values on expected parameters.

[RS-07] If the API detects invalid input values, it MUST return the HTTP status code "`400 Bad Request`". The error payload MUST indicate the erroneous value.

[RS-08] If the API detects syntactically argument names (in the request or query parameters) that are not expected, it SHOULD ignore them.

[RS-09] If the API detects valid values that requires features not implemented, it MUST return the HTTP status code "`501 Not Implemented`". The error payload MUST indicate the unhandled value.

6.4. Resource Model

24. An IP data model should be divided into bounded contexts following a domain-driven design approach. Each bounded context must be mapped to a resource. According to the design principles, a Web API resource model should be decoupled from the data model. A Web API should be modeled as a resource hierarchy to leverage the hierarchical nature of the URI to imply structure (association or composition or aggregation), where each node is either a simple (singleton) resource or a collection of resources. In this hierarchical resource model, the nodes in the root are called top-level nodes and all the rest nested resources are called sub-resources. Sub-resources should be used only to imply aggregations. Moreover, a sub-resource should have strong dependency, i.e. it cannot be top-level resource[1]. Such sub-resources, implying association, are called sub-collections. The other hierarchical structures, i.e. composition and aggregation, should be avoided to avoid complex APIs and duplicate functionality.

25. Only top-level resources, i.e. of maximum one level should be used, else APIs will be complex to be implemented. For example, `https://wipo.int/api/v1/patent?inventorId=12345` should be used instead of `https://wipo.int/api/v1/inventor/12345/patent`.

[RS-10] A Web API SHOULD use only top-level resources. If there are sub-resources, they should be collections and imply association. An entity should be accessible as either top-level resource or sub-resource but not both ways. If a resource can be standalone it MUST be a top-level resource, else a sub-resource.

[RS-11] Query parameters MUST be used instead of URL paths to retrieve nested resources.

[1] Else, we would have multiple ways retrieving the same entities.

26. A Web API should support projection. If only specific attributes from the retrieved data are required, a projection query parameter must be used instead of URL paths. The query parameter should be formed as follows: "`select="<comma-separated list of attribute names>`. A projection query parameter is easier to implement and can retrieve multiple attributes. For example:

```
GET https://wipo.int/api/v1/inventor/id12345?select=firstName,lastName
200 OK
{ …
  "FirstName": "My first name",
  "LastName": "My last name"
}
```

[RS-12] A query parameter SHOULD be used instead of URL paths in case that a Web API supports projection following the format: "`select="<comma-separated list of attribute names>`.

27. There are two types of Web APIs, the CRUD (create, read, update, and delete) Web API and the Intent Web API. CRUD Web APIs model resource create/read/update/delete changes and Intent Web APIs model business operations. CRUD operations should use nouns and Intent Web APIs verbs for the resource names. CRUD Web APIs are the most common but both can be combined for example, the service consumer could use an Intent Web API modeling business operation, which would orchestrate the execution of one or more CRUD Web APIs service operations. Using Intent Web API, the service caller has to orchestrate the business logic but with Intent Web APIs the service provider orchestrates the business logic. CRUD Web APIs are not atomic in comparison to Intent Web APIs[2]. In the case of the CRUD Web API, the resource names should be nouns. In the case of the Intent Web API, the resources names should be verbs.

- For example, a possible business operation is the renewal scenario where the owner of the IP right wants to find the patent and renew it. A CRUD Web API would model this operation in a non-atomic process requiring two actions such as:

```
PUT https://wipo.int/api/v1/patents/id12345
{
  …
}
POST https://wipo.int/api/v1/renewals
{
  …
}
```

- The previous example could be modeled with an atomic service call with an Intent Web API as following:

```
PUT https://wipo.int/api/v1/findAndRenew/id12345
```

[RS-13] The resource names SHOULD be nouns for CRUD Web APIs and verbs for Intent Web APIs.

[RS-14] The noun resource names SHOULD be always named in plural. Irregular noun forms SHOULD NOT be used. For example, /persons should be used instead of /people.

[RS-15] A Web API resource names MUST be composed of words in the English language, using the primary English spellings provided in the Oxford English Dictionary. Resource names that are localized due to business requirements MAY be in other languages.

6.5. Supporting multiple formats

28. Different service consumers may have differing requirements for the data format of the service responses. The media type of the data should be decoupled from the data itself, allowing the service to support a range of media types. Therefore, a Web API must support content type negotiation using the request HTTP header Accept and the response

[2] An Intent API also enables the application of the Command Query Responsibility Segregation (CQRS) pattern. CQRS is a pattern, where you can use a different model to update information than the model you use to read information. The rationale is that for many problems, particularly in more complicated domains, having the same conceptual model for commands and queries leads to a more complex model that is not beneficial.

HTTP header Content-Type as required by IETF RFC 7231. Additionally, a Web API may support other ways of content type negotiation such as query parameter (for example, `?format`), URL suffix (for example `.json`).

29. APIs must support XML and JSON requests and responses following WIPO Standard ST.96. A consistent mapping between these two formats should be used. For the mapping, several conventions are available such as BadgerFish, Parker, Spark, Abdera, JsonML, OData and GData, This Standard recommends the BadgerFish convention due to its simplicity until the JSON schema is provided. HTTP codes should be used among the ones listed in **Error! No bookmark name given.**.

[RS-16] Web APIs MUST support content type negotiation following IETF RFC 7231.

[RS-17] JSON format MUST be assumed when no specific content type is requested.

[RS-18] A Web API SHOULD return the status code "`406 Not Acceptable`" if a requested format is not supported.

[RS-19] A Web API SHOULD reject requests containing unexpected or missing content type headers with the HTTP status code "`406 Not Acceptable`" or "`415 Unsupported Media Type`".

[RS-20] The requests and responses (naming convention, message format, data structure, and data dictionary) SHOULD refer to WIPO Standard ST.96.

[RS-21] A Web API SHOULD provide a consistent mapping between XML and JSON formats for the response and requests.

[RS-22] The names of object properties in JSON SHOULD be LCC, e.g. applicantName while the names of XML components are in UCC according to WIPO Standard ST.96.

6.6. HTTP Methods

30. HTTP Methods (or HTTP Verbs) are a type of function provided by a uniform contract to process resource identifiers and data. HTTP Methods must be used as they were intended to according the standardized semantics as specified in IETF RFC 7231 and 5789, namely:

- GET – retrieve data
- HEAD – like get but without response payload
- POST – submit new data
- PUT – update or create
- PATCH – partial update
- DELETE – delete data
- TRACE – echo
- OPTIONS – query verbs that the server supports for a given URL

31. The uniform contract establishes a set of methods to be used by services within a given collection or inventory. HTTP Methods tunneling may be useful when HTTP Headers are rejected by some firewalls.

[RS-23] HTTP Methods MUST be restricted to the HTTP standard methods POST, GET, PUT, DELETE, OPTIONS, PATCH, TRACE and HEAD as specified in IETF RFC 7231 and 5789.

[RS-24] HTTP Methods MAY follow the pick-and-choose principle, which states that only the functionality needed by the target usage scenario should be implemented.

[RS-25] Some proxies support only POST and GET methods. To overcome these limitations, a Web API MAY use a POST method with a custom HTTP header "tunneling" the real HTTP method.

[RS-26] If a HTTP Method is not supported, the HTTP status code "`405 Method Not Allowed`" SHOULD be returned.

32. In some use cases, multiple operations should be supported at once.

[RS-27] A Web API SHOULD support batching operations (aka bulk operations) instead of multiple individual requests for latency reduction. The same semantics should be used for HTTP Methods and HTTP status codes. If multiple errors occur, the error payload SHOULD contain information about all the occurrences (in the details attribute). All bulk operations SHOULD be executed in an atomic operation.

GET

33. According to IETF RFC 2616, the HTTP protocol does not place any a prior limit on the length of a URI. On the other hand, servers should be cautious about depending on URI lengths above 255 bytes, because some older client or proxy implementations may not properly support these lengths.

[RS-28] If a resource is not found, the method GET MUST return the status code "`404 Not Found`".

[RS-29] If a retrieved successfully, the method GET MUST return 200 OK.

[RS-30] A GET request MUST be idempotent according to the IETF RFC 2616.

[RS-31] When the URI length exceeds the 255 bytes, then the POST method SHOULD be used instead of GET due to GET limitations.

HEAD

34. When a client needs to learn information about an operation, they can use HEAD. HEAD gets the HTTP header you would get if you made a GET request, but without the body. This lets the client determine caching information, what content-type would be returned, what status code would be returned.

[RS-32] A HEAD request MUST be idempotent according to the IETF RFC 2616.

[RS-33] Some proxies support only POST and GET methods. A Web API SHOULD support a custom HTTP request header to override the HTTP Method in order to overcome these limitations.

POST

35. When a client needs to create a resource, they can use POST. For example,

```
POST https://wipo.int/v1/patent
{ "title": "Patent Title" }
Response:
201 Created
Location: https://wipo.int/v1/patent/id12345
{  "id": id12345, "title": "Patent Title" }
```

[RS-34] A POST request MUST NOT be idempotent according to the IETF RFC 2616.

[RS-35] If the resource creation was successful, the HTTP header `Location` SHOULD contain a URI (absolute or relative) pointing to a created resource.

[RS-36] If the resource creation was successful, the response SHOULD contain the status code "`201 Created`".

[RS-37] If the resource creation was successful, the response payload (by default) SHOULD contain the attributes of the created resource, to allow the client to use it without making an additional HTTP call. Consider a custom HTTP header with which the client can suppress the response payload to save bandwidth in case it is not required.

PUT

36. When a client needs to replace an existing resource entirely, they can use PUT. Idempotent characteristics of PUT should be taken into account.

[RS-38] A PUT request MUST be idempotent according to the IETF RFC 2616.

[RS-39] A PUT request has an update semantic (as specified in IETF RFC 7231) and an update or insert semantic. A Web API SHOULD NOT rely on the update or insert semantics of PUT.

[RS-40] If a resource is not found, PUT MUST return the status code "`404 Not Found`".

[RS-41] If a resource is updated successfully, PUT MUST return the status code "`200 OK`" if the updated resource is returned or 204 No Content if it is not returned.

PATCH

37. When a client requires a partial update, they can use PATCH. Idempotent characteristics of PATCH should be taken into account. For example:

```
PATCH https://wipo.int/v1/patent/id12345

If-Match:456

Content-Type: application/merge-patch+json

{ "Title": "Patent Title" }
```

[RS-42] PATCH MUST NOT be idempotent according to the IETF RFC 2616. If a Web API implements partial updates, it SHOULD guarantee idempotent of partial updates. In order to make it idempotent the API MAY follow the IETF RFC 5789 suggestion of using optimistic locking.

[RS-43] If a resource is not found PATCH MUST return the status code "`404 Not Found`".

[RS-44] If a Web API implements partial updates using PATCH), it MUST use the JSON Merge Patch format to describe the partial change set, as described in IETF RFC 7386 (by using the content type `application/merge-patch+json`).

DELETE

38. When a client needs to delete a resource, they can use DELETE.

[RS-45] A DELETE request MUST NOT be idempotent according to the IETF RFC 2616.

[RS-46] If a resource is not found DELETE MUST return the status code "`404 Not Found`".

[RS-47] If a resource is deleted successfully DELETE MUST return the status "`200 OK`" if the deleted resource is returned or `204 No Content` if it is not returned.

TRACE

39. The TRACE method does not carry API semantics and is used for testing and diagnostic information according to IETF RFC 2616, for example for testing a chain of proxies. TRACE allows the client to see what is being received at the other end of the request chain and use that data.

[RS-48] The final recipient is either the origin server or the first proxy or gateway to receive a `Max-Forwards` value of zero in the request. A TRACE request MUST NOT include a body.

[RS-49] A TRACE request MUST NOT be idempotent according to the IETF RFC 2616.

[RS-50] The value of the Via HTTP header field MUST act as a trace of the request chain.

[RS-51] The Max-Forwards HTTP header field MUST be used to allow the client to limit the length of the request chain

[RS-52] If the request is valid, the response SHOULD contain the entire request message in the response body, with a `Content-Type` of "`message/http`".

[RS-53] Responses to TRACE MUST NOT be cached.

[RS-54] The status code "`200 OK`" SHOULD be returned to TRACE.

OPTIONS

40. When a client needs to learn information about a Web API, they can use OPTIONS. OPTIONS do not carry API semantics.

[RS-55] An OPTIONS request MUST be idempotent according to the IETF RFC 2616.Custom HTTP Headers

41. It is a common practice for a Web API using custom HTTP headers and "`X-`" prefix is a common prefix, which RFC 6648 deprecates and discourages to use.

[RS-56] Custom HTTP headers starting with the "`X-`" prefix SHOULD NOT be used.

[RS-57] Custom HTTP headers SHOULD NOT be used to change the behavior of HTTP Methods.

[RS-58] The naming convention for custom HTTP headers is `<organization>-<header name>`, where `<organization>` and `<header>` SHOULD follow the lowercase-dashed convention.

42. According to the design principles, clients and services should evolve independently. Service versioning enables this. Common implementations of service versioning are Header Versioning (by using a custom header), Query string versioning (for example, `?v=v1`), Media type versioning (for example `Accept: application/vnd.v1+json`) and URI versioning (for example `/api/v1/inventor`).

[RS-59] A Web API SHOULD support service versioning. URI versioning SHOULD be used for service versioning such as `/v<version number>` (for example `/api/v1/inventor`). Header Versioning, Query string versioning and Media type versioning SHOULD NOT be used.

43. According to the design principles, service providers and consumers should evolve independently. The service consumer should not be affected from minor backward compatible changes of the service provider. Therefore, service versioning should use only major versions. For internal non-published APIs (for example, for development and testing) minor versions may also be used such as Semantic Versioning.

[RS-60] A versioning-numbering scheme should be followed where only the major service version is used (for example `/v1`).

44. Service endpoint identifiers include information that can change over time. It may not be possible to replace all references to an out-of-date endpoint, which can lead to the service consumer being unable to further interact with the service endpoint. Therefore, the service provider may return a redirection response.

[RS-61] API service contracts MAY Include endpoint redirection feature. When a service consumer attempts to invoke a service, a redirection response may be returned to tell the service consumer to resend the request to a new endpoint. Redirections MAY be temporary or permanent:

- Temporary redirect - using the HTTP response header `Location` and the HTTP status code "`307 Temporary Redirect`" according to IETF RFC 7231; or
- Permanent redirect - using the HTTP response header `Location` and the HTTP status code "`308 Permanent Redirect`" according to IETF RFC 7238.

6.7. Data Query Patterns

Pagination Options

45. Pagination is a mechanism for a client to retrieve data in pages. Using pagination, we prevent overwhelming the service provider with resource demanding requests according to the design principles. The server should enforce a default page size in case the service consumer has not specified one. For example,

[RS-62] A Web API MUST support pagination.

[RS-63] Paginated requests MAY NOT be idempotent, i.e. a paginated request does not create a snapshot of the data.

[RS-64] A Web API MUST use query parameters to implement pagination.

[RS-65] Query parameters `limit=<number of items to deliver>` and `offset=<number of items to skip>` SHOULD be used, where limit is the number of items to be returned (page size), and skip the number of items to be skipped (offset). If no page size limit is specified, a default SHOULD be defined - global or per collection; the default limit MUST be zero "0". A Web API MUST NOT use HTTP headers to implement pagination. For example:

```
https://wipo.int/api/v1/patent?limit=10&offset=20
```

[RS-66] The `limit` and the `offset` parameter values SHOULD be included in the response.

Sorting

46. Retrieving data may require the data to be sorted ascending or descending. Multi-attribute sorting criterion may be used. For example:

```
https://wipo.int/api/v1/patent?orderBy=title asc,creationDate desc
```

[RS-67] A Web API MUST support sorting.

[RS-68] In order to specify a multi-attribute sorting criterion a query parameter MUST be used. The query parameter `orderBy=<comma-separated list of attributes names, attributed with asc or desc>` SHOULD be used. For any attribute name that the order direction is not specified, the default direction MUST be specified.

Expand

47. A service consumer may control the amount of data it receives by expanding a single field into larger objects. Rather than simply asking for a linked entity ID to be included, a service caller can request the full representation of the entity be expanded within the results. Service calls may use expansions to get all the data they need in a single API request. For example:

```
GET https://wipo.int/api/v1/patent?id=id12345&expand=applicant
200 OK
{ "title": “Patent title”, “applicant”={“name”:”applicant name”, …}, …}
In comparison to (if using hypermedia):
GET https://wipo.int/api/v1/patent? id=id12345
200 OK
{ "title": “Patent title”, “applicant”={“href”:”
https://wipo.int/api/v1/link/to/applicant” }, …}
```

[RS-69] A Web API MAY support expanding the body of returned content. The query parameter `expand=<comma-separated list of attributes names>` SHOULD be used.

Number of Elements

48. In some use cases, the consumer of the API may be interested in the number of elements of a collection. This is very common when combined with pagination in order to know the total number of elements of a collection. For example,

```
GET https://wipo.int/api/v1/patent?count=true&top=3&skip=4

200 OK

{ "count": 100, ... }
```

[RS-70] A Web API MUST support returning the number of elements of a collection. The query parameter `count` SHOULD be used.

[RS-71] A Web API MAY support returning the number of elements in a collection inline, i.e. as the part of the response that contains the collection itself. A query parameter MUST be used. The query parameter `count=true` SHOULD be used.

[RS-72] If a Web API supports pagination, it MUST support returning inline in the response the number of the collection (i.e. the total number of elements of the collection).

Complex Search Expressions

49. For retrieving data with only a few search criteria, the query parameters are adequate. If there is a use case where we should search for data using complex search expressions (with multiple criteria, Boolean expressions and search operators) then the API has be to designed more complex and a query language has to be used. A query language has to be supported by a server grammar.

50. The Contextual Query Language (CQL) is a formal language for representing queries to information retrieval systems such as search engines, bibliographic catalogs and museum collection information. Based on the semantics of Z39.50, its design objective is that queries must be readable and writable and that the language is intuitive and maintains the expression of more complex query languages.

[RS-73] A Web API MAY support complex search expressions. A query language SHOULD be specified. CQL MAY be used.

[RS-74] A Service Contract MUST specify the grammar supported (such as fields, functions, keywords, and operators).

[RS-75] The query parameter “q” MUST be used.

6.8. Error Handling

Error Payload

51. Error handling is carried out on two levels: On the protocol level (HTTP) and on the application level (payload returned). Error handling should follow the HTTP standards (RFC 2616). A minimum error payload is recommended, for example:

```
404 Not Found
{
 "error": {
 "code": "03543762",
 "message": "Patent with ID XX not found",
 "target": "/api/v1/patent/id12345",
 "details": [{
 "code": "012312415",
 "message": "Empty result set"
 }]
 }
}
```

[RS-76] On the protocol level, a Web API MUST return an appropriate HTTP status code as described in HTTP Status Codes.

[RS-77] On the application level, a Web API MUST return a payload reporting the error in adequate granularity (mandatory and optional attributes). The following attributes are mandatory:

- `code` - Technical code of the error situation to be used for support purposes
- `message` - User-facing (localizable) message describing the error

The following attributes are conditionally mandatory:

- `details` - If error processing requires nesting of error responses, it MUST use the details field for this purpose. The details field MUST contain an array of JSON objects that shows code and message properties with the same semantics as described above.

The following attributes are optional:

- `target` - The error structure may contain a target attribute that describes a data element (for example, a resource path).
- `status` - Duplicate of the HTTP status code to propagate it along the call chain or to write it in the support log without the need to explicitly add the HTTP status code every time
- `moreInfo` - Array of links containing more information about the error situation, for example, giving hints to the end user
- `internalMessage` - Technical message, for example, for logging purposes

[RS-78] Errors MUST NOT expose security-critical data or internal technical details such as call stacks in the error messages.

[RS-79] The HTTP Header: `Reason-Phrase` (described in RFC 2616) MUST NOT be used to carry error messages.

Correlation ID

52. Typically consuming a service cascades to triggering multiple other services. There should be a mechanism to correlate all the service activations in the same execution context (for example, for including the correlation id in the log messages).

[RS-80] Every logged error SHOULD have a unique Correlation ID. A custom HTTP header SHOULD be used.

6.9. Service Contract

53. REST is not a protocol or an architecture, but an architectural style with architectural properties and architectural constraints; there are no official standards for REST API contracts. This Standard refers to API documentation as a REST Service Contract. The Service Contract is based on the following three fundamental elements:

(a) Resource identifier syntax – how can we express where the data is being transferred to or from?

(b) Methods – what are the protocol mechanisms used to transfer the data?

(c) Media types – what type of data is being transferred? Individual REST services use these elements in different combinations to expose their capabilities. Defining a master set of these elements for use by a collection (or inventory) of services makes this type of service contract "uniform".

[RS-81] A Service Contract format MUST include the following:

- API version;
- Information about the semantics of API elements;
- Resources;
- Resource attributes;
- Query Parameters;
- Methods;
- Media types;
- Search grammar (if one is supported);
- Status Codes;
- HTTP Methods;
- Restrictions and distinctive features;
- Security (if any).

[RS-82] A Service Contract format SHOULD include the following:

- Schemas validating the requests and responses (for example, XSD and JSON Schema);
- Examples of the API usage should be provided in all the supported formats (for example, XML and JSON).

[RS-83] A REST API MUST provide API documentation as a Service Contract.

[RS-84] A Web API implementation deviating from this Standard MUST be explicitly documented in the Service Contract. If a rule that is deviating is not specified in the Service Contract, it MUST be assumed that this Standard is followed.

[RS-85] A Service Contract MUST allow API client skeleton code generation. It SHOULD allow server skeleton code generation.

54. A Web API documentation can be written for example in RESTful API Modeling Language (RAML), Open API Specification (OAS) and WADL. As only RAML fully supports both XML and JSON request/response validation (by using XSD schemas and JSON schemas), this Standard recommends RAML[3].

[RS-86] A Web API documentation SHOULD be written in RAML or OAS. Custom documentation formats SHOULD NOT be used.

[3] OAS is a specification. It also supports Markdown but RAML does not. On the other hand, although both OAS and RAML support JSON Schema validation for the requests and responses, OAS does not support XSDs. Therefore, in the future, when OAS is feature-complete it may be recommended.

6.10. Time-out

55. According to the design principles, the server usage should be limited.

[RS-87] A Web API consumer SHOULD be able to specify a server timeout for each request; a custom HTTP header SHOULD be used. A maximum server timeout SHOULD be also used to protect from over-usage of server resources.

6.11. State Management

56. Following the REST principles, state management must be dealt with on the client side since REST APIs are stateless.

Response Versioning

57. Retrieving multiple times the same data set may result in bandwidth consumption if the data set has not been modified between the requests. Data should be conditionally be retrieved only if it has not been modified.

[RS-88] A Web API SHOULD support conditionally retrieving data. This can be done with Content-based Resource Validation or Time-based Resource Validation. Content-based Resource Validation SHOULD be preferred because it is more accurate.

[RS-89] In order to implement Content-based Resource Validation the ETag HTTP header SHOULD be used in the response to encode the data state. Afterward, this value SHOULD be used in subsequent requests in the conditional HTTP headers (such as If-Match or If-None-Match). If the data has not been modified since the request returned the ETag the server SHOULD return the status code "`304 Not Modified`" (if not modified). This mechanism is specified in IETF RFC 7231 and 7232.

[RS-90] In order to implement Time-based Resource Validation the Last-Modified HTTP header SHOULD be used. This mechanism is specified in IETF RFC 7231 and 7232.

[RS-91] Using response versioning a service consumer MAY implement Optimistic Locking.

Caching

58. A Web API implementation should support cache handling for bandwidth saving in compliance with the IETF RFC 7234.

[RS-92] A Web API MUST support caching of GET results; a Web API MAY support caching of results from other HTTP Methods.

[RS-93] The HTTP response headers `Cache-Control` and `Expires` SHOULD be used. The latter MAY be used to support legacy clients.

Managed File Transfer

59. Transferring (i.e. downloading or uploading) large files has a high probability of causing a network interruption or some other transmission failure. It also requires a large amount of memory in the service provider and service consumer. Therefore, it is recommended to transfer large files in multiple chunks and multiple requests. This option also provides an indication of the total download or upload progress. The partial transferring of large files should resume support. The service provider should advertise if it supports the partial transferring of large files.[4]

[4] The service provider may return the location of the file and then the service consumer call a directory service to download the file. At the end, partial file download is required. This paragraph does not take into account non-REST protocols such as FTP or sFTP or rsync.

60. There are two approaches for transferring chunks; one is using the `Transfer-Encoding: chunked` header and the other using the `Content-Length` header. These headers should not be used together. `Content-Length` indicates the full size of the file transferred, and therefore the receiver will know the length of the body and will be able to estimate the download completion time. The `Transfer-Encoding: chunked` header is useful for streaming infinitely bounded data, such as audio or video, but not files. It is recommended to use the `Content-Length` header for downloading as the server utilization is low in comparison to `Transfer-Encoding: chunked.` For uploading, the `Transfer-Encoding: chunked` header is recommended.

- A Web API should advertise if it supports partial file downloads by responding to `HEAD` requests and replying with the HTTP response headers `Accept-Ranges` and `Content-Length`. The former should indicate the unit that can be used to define a range and should not be none. The latter indicates the full size of the file to download.

[RS-94] A Web API SHOULD advertise if it supports partial file downloads by responding to `HEAD` requests and replying with the HTTP response headers `Accept-Ranges` and `Content-Length`.

61. A Web API that supports downloading large files should support partial requests according to IETF RFC 7232, i.e.:

- The service consumer asking for a range should use the HTTP header `Range`.
- The service provider response should contain the HTTP headers `Content-Range` and `Content-Length`.
- The service provider response should have the HTTP status `206 Partial Content` in case of a successful range request. In case of a range request that is out of bounds (range values overlap the extent of the resource), the server responds with a `416 Requested Range Not Satisfiable` status. In case of no support of range requests, the `200 OK` status is sent back from a server.

[RS-95] A Web API SHOULD support partial file downloads. Multipart ranges SHOULD be supported.

62. Multipart ranges may also be requested if the HTTP header `Content-Type: multipart/byteranges; boundary=XXXXX` is used. A range request may be conditional if it is combined with `ETag` or `If-Range` HTTP Headers.

63. There is not any IETF RFC for large files upload. Therefore, in the Standard we do not provide any implementation recommendation for its implementation.

[RS-96] A Web API SHOULD advertise if it supports partial file uploads.

[RS-97] A Web API SHOULD support partial file uploaded. Multipart ranges SHOULD be supported.

64. The IETF RFC 2616 does not impose any specific size limit for requests. The API Service Contract should specify the maximum limit for the requests. Moreover, on runtime the service producer should indicate to the service consumer if the allowed maximum limit has been exceeded.

[RS-98] The service provider SHOULD return with HTTP response headers the HTTP header `413 Request Entity Too Large` in case the request has exceeded the maximum allowed limit. A custom HTTP header MAY be used to indicate the maximum size of the request.

6.12. Preference Handling

65. A service provider may allow a service consumer to configure values and influence how the former processes the requests of the latter.

[RS-99] If a Web API supports preference handling it SHOULD be implemented according to IETF RFC 7240, i.e. the request HTTP header `Prefer` SHOULD be used and the response HTTP header `Preference-Applied` should be returned (echoing the original request). The nomenclature of preferences that may be set by using the Prefer header MUST be recorded in the Service Contract.

6.13. Translation

66. A service consumer may request responses in a specific language if the service provider supports it.

[RS-100] If a Web API supports localized data, the request HTTP header `Accept-Language` MUST be supported to indicate the set of natural languages that are preferred in the response as specified in IETF RFC 7231.

6.14. Long-Running Operations

67. There are cases, where it involves long running operations. For instance, the generation of a PDF in the service provider may take some minutes. This paragraph recommends a typical message exchange pattern to implement such cases, for example:

```
// (a)
GET https://wipo.int/api/v1/patent
Content-Type: application/pdf
…
// (b)
HTTP/1.1 202 Accepted
Location: https://wipo.int/api/v1/queue/12345
…
// (c1)
GET https://wipo.int/api/v1/queue/12345
…
HTTP/1.1 200 OK
…
// (c2)
GET https://wipo.int/api/v1/queue/12345
HTTP/1.1 303 See Other
Location: https://wipo.int/api/v1/path/to/pdf
…
// (c3)
GET https://wipo.int/api/v1/path/to/pdf
…
```

[RS-101] If the API supports long-running operations, they SHOULD be asynchronous. The following approach SHOULD be followed:

(a) The service consumer activates the service operation.

(b) The service operation returns the status code "`202 Accepted`" according to IETF RFC 7231 (section 6.3.3), i.e. the request has been accepted for processing but the processing has not been completed. The location of the queued task that was created is also returned with the HTTP header Location.

(c) The service consumer calls the returned location to learn if the resource is available. If the resource is not available, the response SHOULD have the status code "`200 OK`", contain the task status (for example pending) and MAY contain other information (for example, a link to cancel the task using the DELETE HTTP method). If the resource is available, the response SHOULD have the status code "`303 See Other`" and the HTTP header Location SHOULD contain the URL to retrieve the task results.

6.15. Security Model

General Rules

68. Within the scope of this standard, API security is concerned with pivotal security attributes that will ensure that information and APIs are secure throughout their lifecycle. These attributes are confidentiality, integrity, availability, trust, non-repudiation, compartmentalization, authentication, authorization and auditing.

[RS-102] Confidentiality**:** APIs and Information MUST be identified, classified, and protected against unauthorized access, disclosure and eavesdropping at all times. The least privilege, need to know and need to share principles MUST be followed accordingly.

[RS-103] Integrity-Assurance: APIs and Information MUST be protected against unauthorized modification, duplication, corruption and destruction. Information MUST be modified through approved transactions and interfaces. Systems MUST be changed through approved configuration management, change management and patch management processes.

[RS-104] Availability: APIs and Information MUST be available to authorized users at the right time as defined in the SLAs, access-control policies and defined business processes.

[RS-105] Non-repudiation: Every transaction processed or action performed by APIs MUST enforce non-repudiation through the implementation of proper auditing, authorization, authentication, and the implementation of secure paths and non-repudiation services and mechanisms.

[RS-106] Authentication, Authorization, Auditing: Users, systems, APIs or devices involved in critical transactions or actions MUST be authenticated, authorized using role-based or attribute based access-control services and maintain segregation of duty. In addition, all actions MUST be logged and the authentication's strength must increase with the associated information risk.

Guidelines for secure and threat-resistant API management

69. APIs should be designed, built, tested, and implemented with security requirements and risks in mind. The appropriate countermeasures and controls should be built directly into the design and not as an after-thought. It is recommended to use best practices and standards such as OWASP.

[RS-107] While developing APIs, threats, malicious use cases, secure coding techniques, transport layer security and security testing MUST be carefully considered, especially:

- PUTs and POSTs – i.e.: which change internal data and could be potentially used to attack or misinform
- DELETES – i.e.: which could be used to remove the contents of an internal resource repository
- Whitelist allowable methods- to ensure that allowable HTTP Methods are properly restricted while others would return a proper response code
- Well known attacks should be considered during the threat-modeling phase of the design to ensure that the threat surface does not increase. The threats and mitigation defined within OWASP Top Ten Cheat Sheet MUST be taken into consideration.
- The standards and best practices listed below SHOULD be followed:
    - Secure coding best practices: OWASP Secure Coding Principles
    - Rest API security: REST Security Cheat Sheet
    - Escape inputs and cross site scripting protection: OWASP XSS Cheat Sheet
    - SQL Injection prevention: OWASP SQL Injection Cheat Sheet, OWASP Parameterization Cheat Sheet
    - Transport layer security: OWASP Transport Layer Protection Cheat Sheet

[RS-108] Security testing and vulnerability assessment MUST be carried out to ensure that APIs are secure and threat-resistant. This requirement can be achieved by leveraging Static and Dynamic Application Security Testing (SAST/DAST), automated vulnerability management tools and penetration testing.

Encryption, Integrity and non-repudiation

Protected services MUST be secured to protect authentication credentials in transit: for example passwords, API keys or JSON Web Tokens. Integrity of the transmitted data and non-repudiation of action taken SHOULD also be guaranteed. Secure cryptographic mechanisms can ensure confidentiality, encryption, integrity assurance and non-repudiation

[RS-109] Protected services MUST only provide HTTPS endpoints. TLS 1.2 or higher with a cipher suite that include ECDHE for key exchange and perfect forward secrecy SHOULD be used to provide transport security. The use of insecure cryptographic algorithms and backwards compatibility to SSL 3 and TLS 1.0/1.1 SHOULD NOT be allowed.

[RS-110] For maximum security and trust, a site-to-site IPSEC VPN SHOULD be established to further protect the information that is going over insecure networks.

[RS-111] The consuming application SHOULD validate the TLS certificate chain when making requests to protected resources, including checking the certificate revocation list.

[RS-112] Protected services SHOULD only use valid certificates issued by a trusted certificate authority (CA).

[RS-113] Tokens SHOULD be signed using secure signing algorithms that are compliant with the digital signature standard (DSS) FIPS –186-4. The RSA digital signature algorithm or the ECDSA algorithm SHOULD be considered.

Authentication and Authorization

70. Authorization is the act of performing access control on a resource. Authorization does not just cover the enforcement of access controls, but also the definition of those controls. This includes the access rules and policies, which should define the required level of access agreeable to both provider and consuming application. The foundation of access control is a provider granting or denying a consuming application and/or consumer access to a resource to a certain level of granularity. Coarse-grained access should be considered at the API or the API gateway request point while fine-grained control should be considered at the backend service if possible. Role Based Access Control (RBAC) or the Attribute Based Access Control (ABAC) model can be considered.

71. If a service is protected, then Open ID Connect should be favored over OAuth 2.0 because the former fills many of the gaps of the latter such as a standardized way to gain a resource owner's profile data, JSON Web Token (JWT) standardized token format and cryptography. Other security schemes should not be used such as HTTP Basic Authorization which the client needs to keep the password somewhere in clear text to send it along with each request and the verification of a password should be slower because it will have to access the credential store. OAuth 2.0 supports does not specify the security token. Therefore, the JWT token should be used in comparison for example to SAML 2.0, which is more verbose.

[RS-114] Anonymous authentication MUST only be used when the customers and the application they are using accesses information or feature with a low sensitivity level therefore not requiring authentication in anyway i.e.: public information.

[RS-115] Username and password or password hash authentication MUST NOT be allowed.

[RS-116] If a service is protected, then Open ID Connect SHOULD be used. Alternatively, OAuth 2.0 JWT security token SHOULD be used.

[RS-117] JWT (JSON Web Token considerations):

- JWT secret MUST have high entropy to increase the work factor of token brute force
- Token TTL and RTTL SHOULD be as short as possible.
- Sensitive information SHOULD not be stored in the JWT payload

72. A common security design choice is if user authentication should be centralized in an Identity Provider (IdP) or local.

[RS-118] User authentication SHOULD be centralized in an IdP, which issues access tokens.

73. Services should be careful to prevent leaking credentials. Passwords, security tokens, and API keys should not appear in the URL, as this can be captured in web server logs, which makes them intrinsically valuable. For example, the following is incorrect (API Key in URL): `https://wipo.int/api/patent?apiKey=a53f435643de32`

[RS-119] In POST/PUT requests, sensitive data SHOULD be transferred in the request body or request headers.

[RS-120] In GET requests, sensitive data SHOULD be transferred in an HTTP Header.

[RS-121] In order to minimize latency and reduce coupling between protected services, the access control decision SHOULD be taken locally by REST endpoints.

74. API Keys Authentication: API keys should be used wherever system-to-system authentication is required API keys should automatically and randomly generated. The inherent risk of this authentication mode is that anyone with a copy of the API key can use it as though they were the legitimate consuming application. Hence, all communications should be over TLS, to protect the key in transit. The onus is on the application developer to properly protect their copy of the API key. If the API key is embedded into the consuming application, it can be decompiled and extracted. If stored in plain text files they can be stolen and re-used for malicious purposes. API Key must therefore be protected by a credential store or a secret management mechanism. API Keys may be used to control services usage even for public services.

[RS-122] API Keys SHOULD be used for protected and public services to prevent overwhelming their service provider with multiple requests (denial-of-service attacks). For protected services API Keys MAY be used for monetization (purchased plans), usage policy enforcement (QoS) and monitoring.

[RS-123] API Keys MAY be combined with the HTTP request header user-agent to discern between a human user and a software agent as specified in IETF RFC 7231.

[RS-124] The service provider SHOULD return with HTTP response headers the current usage status. The following response attributes MAY be returned:

- rate limit - rate limit (per minute) as set in the system;
- rate limit remaining - remaining amount of requests allowed during the current time slot (-1 indicates that the limit has been exceeded);
- rate limit reset - time (in seconds) remaining until the request counter will be reset.

[RS-125] The service provider SHOULD return the status code "`429 Too Many Requests`" if requests are coming in too quickly.

[RS-126] API Keys MUST be revoked if the client violates the usage agreement.

[RS-127] API Keys SHOULD be transferred using custom HTTP headers. They SHOULD NOT be transferred using Query Parameters.

[RS-128] API Keys SHOULD be randomly generated

Certificate mutual authentication should be used when a Web API requires stronger authentication than offered by API keys and therefore overhead of public key cryptography and certificate are warranted. Secure and trusted certificates must be issued by a mutually trusted certificate authority (CA) through a trust establishment process or cross-certification.

[RS-129] For highly privileged services, 2-way mutual authentication between the client and the server SHOULD be used using certificates to provide additional protection (for example, X.509).

To mitigate identity security risks peculiar to sensitive systems and privileged actions, strong authentication can be leveraged.

[RS-130] Multi-factor authentication SHOULD be implemented to mitigate identity risks for application with high risk profile, system processing very sensitive information or privileged action.

Availability and threat protection

75. Availability in this context covers threat protection to minimize API downtime, looking at how threats against exposed APIs can be mitigated using basic design principles. Availability also covers scaling to meet demand and ensuring the hosting environments are stable etc. These levels of availability are addressed across the hardware and software stacks that support the delivery of APIs. Availability is normally addressed under business continuity and disaster recovery standards that recommend a risk assessment approach to define the availability requirements.

[RS-131] Distributed Denial of Service (DDOS) and threat protection SHOULD be provided by:

- Throttling access to all exposed APIs
- Performing egress and egress filtering
- Leveraging the implementation of perimeter security controls such as next generation firewalls, load balancers, DDOS protection systems and services, and Web Application Firewalls
- Active threat monitoring leveraging SIEM for event management and correlation
- Leveraging the security gateway native capabilities such as throttling and white-listing
- Definition and implementation of SLAs

[RS-132] Ensure that business continuity and disaster recovery, processes and actionable plans are in place and tested

Cross-domain Requests

76. Certain "cross-domain" requests, notably Ajax requests, are forbidden by default by the same-origin security policy. Under the same-origin policy, a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin (i.e. combination of URI scheme, host name, and port number).

77. The Cross-Origin Resource Sharing (CORS) is a W3C standard to flexibly specify which Cross-Domain Requests are permitted. By delivering appropriate CORS HTTP headers, your REST API signals to the browser which domains or origins are allowed to make JavaScript calls to the REST service.

78. The JSON with padding (JSONP) is a method for sending JSON data without worrying about cross-domain request issues. It introduces callback functions for the loading of JSON data from different domains. The idea behind it is based on the fact that the HTML `<script>` tag is not affected by the same origin policy. Anything imported through this tag is executed immediately in the global context. Instead of passing in a JavaScript file, one can pass in a URL to a service that returns JavaScript code.

79. The following approaches are usually followed to bypass this restriction:

- JSONP is a workaround for cross-domain requests. It does not offer any error-detection mechanism, i.e. if anything went wrong and the service failed or responded with an HTTP error, there is no way to find out what happened on the client side; the AJAX application will just hang. Moreover, the site that uses JSONP blindly trusts the served JSON from a different domain.
- Iframe is a workaround for cross-domain requests. Using the JavaScript `window.postMessage(message, targetOrigin)` method on the iframe object, it is possible to pass a request a site of a different domain. Iframe approach has good compatibility even in old browsers. Moreover, it only supports GET. The source of the iframes page should be always be checked due to security issues.
- CORS is a standardized approach to perform a call to an external domain. It can use `XMLHttpRequest` to send and receive data and has better error handling mechanism than JSONP. It supports many types of authorization in comparison to JSONP, which only supports cookies. It also supports HTTP Methods in comparison to JSONP, which only supports GET. On the other hand, it is not always possible to implement CORS because the browsers have to support it and because the API consumers have to be enlisted in the CORS whitelist.

[RS-133] If the REST API is public then the HTTP header `Access-Control-Allow-Origin` MUST be set to '*'.

[RS-134] If the REST API is protected then CORS SHOULD be used if possible. Else, JSONP SHOULD be used as fallback (but only for GET requests). Iframe SHOULD NOT be used.

6.16. API Maturity Model

80. It is common classifying a REST API using a maturity model. While various models are available, this Standard refers to the Richardson Maturity Model (RMM). RMM defines three levels and this Standard recommends Level 2 for REST API because Level 3 is complex to implement and requires significant conceptual and development-related investment from service providers and consumers. At the same time, it does not immediately benefit service consumers.

81. If a Web API implements level 3 of RMM, a hypermedia format must be put in place. HAL should be used because it is simple and because it is compatible with JSON and XML responses. Other hypermedia formats should not be used such as:

- IETF RFC 5988 – it defines the semantics of Web linking in a serialization-agnostic way, while setting the focus on plain text serialization as content for the HTTP header Link
- JSON-LD – for augmenting existing API JSON responses
- Collection+JSON – is a full featured media type for JSON responses

[RS-135] A Web API MUST implement at least level 2 (Transport Native Properties) of RMM. Level 3 (Hypermedia) MAY be implemented to make the API completely discoverable.

82. A custom hypermedia format may be designed. In this case, a set of attributes is recommended, for example:

```
{
   "link": {
      "href": "/patent",
      "rel": "self"
   },
   ...
 }
```

[RS-136] A custom hypermedia format MAY be designed. The following set of attributes SHOULD be used enclosed into an attribute link:

- `href` – the target URI
- `rel` – the meaning of the target URI
- `self` – the URI references the resource itself
- `next` – the URI references the previous page (if used during pagination)
- `previous` – the URI references the next page (if used during pagination)
- arbitrary name v denotes the custom meaning of a relation

7. SOAP WEB API

83. A SOAP Web API is a software application identified by URI, whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts, and supports direct interactions with other software applications using XML-based messages via internet protocols such as SOAP and HTTP.

84. A SOAP-based contract is described in a Web Service Definition Language (WSDL), a W3C standard, document. Further, in this document “Web Service Contract WSDL document” will be referred as just “WSDL”.

85. When creating web services, there are two development styles: Contract Last and Contract First. When using a contract-last approach, you start with the code, and let the web service contract be generated from that. When using contract-first, you start with the WSDL contract, and use code to implement said contract.

7.1. General Rules

86. The Web Service Interoperability (WS-I) Profile is one of the most important standards, which provides a minimum foundation for writing Web Services that can work together. WS-I provides a guideline on how services are “exposed” to each other and how they transfer information (messaging). It is a profile for implementing specific versions of some of the most important Web Service standards such as WSDL, SOAP, XML, etc. Adhering to certain profiles implicitly indicates adhering to specific versions of these Web Services standards. WS-I Basic Profile v1.1 provides guidance for using XML 1.0, HTTP 1.1, UDDI, SOAP 1.1, WSDL 1.1, and UDDI 2.0. WS-I Basic Profile 2.0 provides guidance for using SOAP 1.2, WSDL 1.1, UDDI 2.0, WS-Addressing, and MTOM. SOAP 1.2 provides a clear processing model and leads to better interoperability. WSDL 2.0 was designed to solve the interoperability issues found in WSDL 1.1 by using improved SOAP 1.2 bindings.

[WS-01] All WSDLs MUST conform to WS-I Basic Profile 2.0. WSDL 1.2 MAY be used.

87. A WSDL SOAP binding can be either a Remote Procedure Call (RPC) style binding or a document style binding. A SOAP binding can also have an encoded use or a literal use. This gives you five style/use models: RPC/encoded, RPC/literal, document/encoded, document/literal, document/literal wrapped.

[WS-02] Services MUST follow document style binding and literal use model (document/literal or document/literal wrapped). Exceptional use cases are when in the WSDL there are overloaded operations (all the other styles SHOULD be used) or when there are graphs (the RPC/encoded style MUST be used).

88. The concrete WSDL should be separated from the abstract WSDL in order to provide a more modular and flexible interface. The abstract WSDL defines data types, messages, operation, and the port type. The concrete WSDL defines the binding, port and service.

[WS-03] The WSDL SHOULD be separated into an abstract and a concrete part.

[WS-04] All data types SHOULD be defined in an XSD file and imported in the abstract WSDL.

[WS-05] The concrete WSDL MUST define only one service with one port.

7.2. Schemas

89. Schemas used in the WSDL must follow the WIPO Standard ST.96 Standard. For re-use purposes and modularity, a schema must be a separate document that is included or imported into the WSDL instead of defining directly it in the WSDL. This will permit changes in XML structure without changing the WSDL.

[WS-06] The schema defined in the `wsdl:types` element MUST be imported from a self-standing schema file, to allow modularity and re-use.

[WS-07] Import of an external schema MUST be implemented using an `xsd:import` technique, not an `xsd:include`.

[WS-08] Element `xsd:any` MUST NOT be used to specify a root element in the message body.

[WS-09] The target namespace for the WSDL (attribute `targetNamespace` on `wsdl:definitions`) MUST be different from the target namespace of the schema (attribute `targetNamespace` on `xsd:schema`).

[WS-10] The requests and responses (naming convention, message format, data structure, and data dictionary) SHOULD follow WIPO Standard ST.96.

7.3. Naming and Versioning

[WS-11] Services MUST be named in UpperCamelCase and have a 'Service' suffix.

[WS-12] WSDL elements message, part, portType, operation, input, output, and binding SHOULD be named in UpperCamelCase.

[WS-13] Request message names SHOULD have a ‘Request’ suffix.

[WS-14] Response message names SHOULD have a ‘Response` suffix.

[WS-15] Operation names SHOULD follow the format of `<Verb><Object>{<Qualifier>}`, where `<Verb>` indicates the operation (preferably Get, Create, Update, or Delete where applicable) on the `<Object>` of the operation, optionally finally followed by a `<Qualifier>` of the `<Object>`.

90. All operation names will have at least two parts. An optional third part may be included to further clarify and/or specify the business purpose of the operation. The three parts are: `<Verb> <Object> <Qualifier - Optional>`. Each part will be described in detail below.

Verb – Each operation name will start with a verb. The examples of verb usage in most commonly used operations are described below:

Verb	Description	Example
Get	Get a single object	`GetBibData`
Create	Get a new object	`CreateBibData`
Update	Update an object	`UpdateBibData`
Delete	Delete an object	`DeleteCustomer`

Object – The noun following the verb will be a succinct and unambiguous description of the business function the operation is providing. The goal is to provide consumers with a better understanding of what the operation does with no ambiguity. Given that the definition of some entities are not common across the various cost centers, the object may be a composite field with the first node being the cost center and the second node the entity, for example, `PatentCustomer`.

Qualifier – The purpose of the object qualifier optional attribute is, to further elucidate the business domain or subject area, for example, `GetCustomerList`. Get denotes the operation to be acted upon the Customer and List further describes the fact that the intention is to get a list of Customers not just one customer as in `GetCustomer`.

91. According to the design principles, service providers and consumers should evolve independently. The service consumer should not be affected from minor backward compatible changes of the service provider. Therefore, service versioning should use only major versions. For internal APIs (for example, for development and testing) minor versions may also be used such as Semantic Versioning.

[WS-16] The name of the WSDL file SHOULD conform the following pattern: `<service name>_V<major version number>`

[WS-17] The namespace of the WSDL file SHOULD contain the service version; for example `https://wipo.int/PatentsService/V1`"

92. The description of service and its operations is provided as WSDL documentation.

[WS-18] Element `wsdl:documentation` SHOULD be used in WSDL with description of service (as the first child of `wsdl:definitions` in the WSDL) and its operations.

7.4. Web Service Contract Design

93. A Web Service contract should include a technical interface comprised of a Web Service Definition Language (WSDL), XML Schema definitions, WS-Policy descriptions as well as a non-technical interface comprised of one or more service description documents.

94. The WSDL, part of the “Service Contract,” must be designed prior to any code development. No WSDL should ever be auto-generated from the code. The motto is “Contract First” and NOT “Code First”. All Web Service Contract must conform to Web Service Interoperability Basic Profile (WS-I BP). Any project that auto-generates from code will be liable to go back and make all required changes to ensure conformance to these standards.

7.5. Attaching Policies to WSDL Definitions

95. Web service contracts can be extended with security policies that express additional constraints, requirements, and qualities that typically relate to the behaviors of services. Security policies can be human-readable and become part of a supplemental service-level agreement, or can be machine-readable processed at runtime. Machine-readable policies are defined using the WS-Policy language and related WS-Policy specifications.

[WS-19] Policy expressions MUST be isolated into a separate WS-Policy definition document, which is then referenced within the WSDL document via the `wsp:PolicyReference` element.

[WS-20] Global or domain-specific policies SHOULD be isolated and applied to multiple services.

[WS-21] Policy attachment points SHOULD conform the WSDL 1.1 or later version, preferably version 2.0, attachment point elements and corresponding policy subjects (service, endpoint, operation, and message).

7.6. SOAP – Web Service Security

96. Web Services Security (WSS): SOAP Message Security is a set of enhancements to SOAP messaging that provides message integrity and confidentiality. WSS: SOAP Message Security is extensible, and can accommodate a variety of security models and encryption technologies. WSS: SOAP Message Security provides three main mechanisms that can be used independently or together:

- The ability to send security tokens as part of a message, and for associating the security tokens with message content
- The ability to protect the contents of a message from unauthorized and undetected modification (message integrity)
- The ability to protect the contents of a message from unauthorized disclosure (message confidentiality)

WSS: SOAP Message Security can be used in conjunction with other Web service extensions and application-specific protocols to satisfy a variety of security requirements.

[WS-22] Web Services using SOAP message SHOULD be protected accordance with WSS:SOAP Standard recommendations.

8. DATA TYPE FORMATS

97. This Standard recommends primitive data type formats such as time, date and language to be consistent with the recommendation of WIPO Standard ST.96.

[CS-01] Time MUST be formatted as specified in IETF RFC 3339 (it is a profile of ISO 8601). Time zone information SHOULD be used as specified in IETF RFC 3339. For example: `20:54:21+00:00`

[CS-02] Date MUST be formatted as specified in IETF RFC 3339 (it is a profile of ISO 8601). For example: `2018-10-19`

[CS-03] Datetime (i.e. timestamp) MUST be formatted as specified in IETF RFC 3339 (it is a profile of ISO 8601). Time zone SHOULD be used as specified in IETF RFC 3339. For example: `2017-02-14T20:54:21+00:00`

[CS-04] ISO 4217-Alpha (3-Letter Currency Codes) MUST be used for Currency Codes. The precision of the value (i.e. number of digits after the decimal point) MAY vary depending on the business requirements.

[CS-05] WIPO Standard ST.3 two-letter codes MUST be used for representing IPOs and for priority and designated country/organization.

[CS-06] ISO 3166-1-Alpha-2 Code Elements (2 letter country codes) MUST be used for the representation of the names of countries, dependencies, and other areas of particular geopolitical interest, on the basis of lists of country names obtained from the United Nations.

[CS-07] ISO 639-1 (2-Letter Language Codes) MUST be used for Language Codes.

[CS-08] Units of Measure SHOULD use the units of measure as described in The Unified Code for Units of Measure (based on ISO 80000 definitions). For example: `kg`

9. CONFORMANCE

98. This Standard is designed as a set of rules and conventions that can be layered on top of existing or new Web Service APIs to provide common functionality. Not all services will support all of the conventions defined in the Standard due to business (for example, QoS may not be required) or technical constraints (for example, OAuth 2.0 may already be used).

99. This Standard defines three levels of conformance: Minimal, Intermediate and Advanced Conformance Levels.

100. The Web Service APIs are encouraged to support as much additional functionality beyond their level of conformance as is appropriate for their intended scenario.

101. Three conformance levels are defined:

- **Level A**: For Level A conformance (the minimum level of conformance), the API indicates that the required design rules are followed.
- **Level AA:** For Level AA conformance, the API indicates that is Level A compliant and all the recommended design rules are followed.
- **Level AAA:** For Level AAA conformance, the API indicates that is Level AA compliant and all the possible design rules are followed.

102. The traceability matric between the design rules and the Conformance Levels is listed in Annex I.

7. REFERENCES

Standards and Conventions

- IEFT RFC 2119: Key words for use in RFCs to Indicate Requirement Levels – www.ietf.org/rfc/rfc2119.txt
- IEFT RFC 3339: Date and Time on the Internet: Timestamps – www.ietf.org/rfc/rfc3339.txt
- IEFT RFC 3986: Uniform Resource Identifier (URI): Generic Syntax – www.ietf.org/rfc/rfc3986.txt
- IEFT RFC 5789: PATCH Method for HTTP – https://tools.ietf.org/rfc/rfc5789.txt
- IEFT RFC 5988: Web Linking – https://tools.ietf.org/rfc/rfc5988.txt
- IEFT RFC 6648: Deprecating the "X-" Prefix and Similar Constructs in Application Protocols – https://tools.ietf.org/rfc/rfc6648.txt
- IEFT RFC 6750: The OAuth 2.0 Authorization Framework: Bearer Token Usage – https://tools.ietf.org/rfc/rfc6750.txt
- IEFT RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content – www.ietf.org/rfc/rfc7231.txt
- IEFT RFC 7232: Hypertext Transfer Protocol (HTTP/1.1) – Conditional Requests www.ietf.org/rfc/rfc7232.txt
- IEFT RFC 7234: Hypertext Transfer Protocol (HTTP/1.1) – Caching www.ietf.org/rfc/rfc7234.txt
- IEFT RFC 7386: JSON Merge Patch – www.ietf.org/rfc/rfc7386.txt.
- IEFT RFC 7240: Prefer Header for HTTP – https://tools.ietf.org/rfc/rfc7240.txt
- IEFT RFC 7519: JSON Web Token – www.ietf.org/rfc/rfc7519.txt
- IEFT RFC 7540: Hypertext Transfer Protocol Version 2 (HTTP/2) – https://tools.ietf.org/html/rfc7540
- IEFT BCP-47: Tags for Identifying Languages – https://tools.ietf.org/rfc/bcp/bcp47.txt.
- ISO 639-1: Language codes – https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
- ISO 3166-1 alpha-2: Two-letter acronyms for country codes – https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2
- ISO 3166-1 alpha-3: Three-letter acronyms for country codes – https://en.wikipedia.org/wiki/ISO_3166-1_alpha-3
- ISO 4217: Currency Codes – www.iso.org/iso/home/standards/currency_codes.htm
- ISO 8601: Date and Time Formats – https://en.wikipedia.org/wiki/ISO_8601
- OData
- OASIS OData Metadata Service Entity Model – http://docs.oasis-open.org/odata/odata/v4.0/os/models/MetadataService.edmx.
- OASIS OData JSON Format Version 4.0. Edited by Ralf Handl, Michael Pizzo, and Mark Biamonte. Latest version – http://docs.oasis-open.org/odata/odata-json-format/v4.0/odata-json-format-v4.0.html.
- OASIS OData Atom Format Version 4.0. Edited by Martin Zurmuehl, Michael Pizzo, and Ralf Handl. Latest version – http://docs.oasis-open.org/odata/odata-atom-format/v4.0/odata-atom-format-v4.0.html.
- OASIS OData "OData Version 4.0 Part 1: Protocol– http://docs.oasis-open.org/odata/odata/v4.0/os/part1-protocol/odata-v4.0-os-part1-protocol.html.
- OASIS OData Version 4.0 Part 2: URL Conventions – http://docs.oasis-open.org/odata/odata/v4.0/os/part2-url-conventions/odata-v4.0-os-part2-url-conventions.html.
- OASIS OData Version 4.0 Part 3: Common Schema Definition Language (CSDL) – http://docs.oasis-open.org/odata/odata/v4.0/os/part3-csdl/odata-v4.0-os-part3-csdl.html.
- OASIS ABNF components: OData ABNF Construction Rules Version 4.0 and OData ABNF Test Cases – http://docs.oasis-open.org/odata/odata/v4.0/os/abnf/
- OASIS Vocabulary components: OData Core Vocabulary, OData Measures Vocabulary and OData Capabilities Vocabulary – http://docs.oasis-open.org/odata/odata/v4.0/os/vocabularies/
- OASIS XML schemas: OData EDMX XML Schema and OData EDM XML Schema– http://docs.oasis-open.org/odata/odata/v4.0/os/schemas/
- OASIS SAML 2.0 – http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html
- RAML (RESTful API Modeling Language) – http://raml.org
- OpenAPI Initiative – www.openapis.org
- Richardson's REST API Maturity Model – https://martinfowler.com/articles/richardsonMaturityModel.html
- HAL – http://stateless.co/hal_specification.html
- JSON-LD – https://json-ld.org
- Collection+JSON - Document Format – http://amundsen.com/media-types/collection/format/
- BadgerFish – http://badgerfish.ning.com/
- Semantic Versioning – https://semver.org/
- REST – https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- CQL – https://en.wikipedia.org/wiki/Contextual_Query_Language
- Z39.50 – https://www.loc.gov/z3950/agency/Z39-50-2003.pdf
- WS-I Basic Profile 2.0 – http://ws-i.org/profiles/basicprofile-2.0-2010-11-09.html
- W3C SOAP 1.2 Part 1: Messaging Framework – https://www.w3.org/TR/soap12-part1/
- W3C SOAP 1.2 Part 2: Adjuncts – https://www.w3.org/TR/soap12-part2/
- W3C WSDL Version 2.0 Part 1: Core Language – https://www.w3.org/TR/wsdl20/
- W3C CORS - https://www.w3.org/TR/cors/

- W3C Matric Parameters – https://www.w3.org/DesignIssues/MatrixURIs.html

IP Offices' REST APIs

- EPO – Open Patent Services OPS v 3.2 https://developers.epo.org
- USPTO – PatentsView http://www.patentsview.org/api/doc.html
- WIPO – ePCTv1.1 https://pct.wipo.int/
- EUIPO – TMview, Designview, TMclass http://www.tm-xml.org/TM-XML/TM-XML_xml/TM-XML_TM-Search.xml

Industry REST APIs and Design Guidelines

- Facebook – https://developers.facebook.com/docs/graph-api/reference
- GitHub – https://developer.github.com/v3
- Google APIs Design Guide – https://cloud.google.com/apis/design/
- Azure – https://docs.microsoft.com/en-us/rest/api/
- OpenAPI – https://swagger.io/docs/specification/about/
- OData – http://www.odata.org/documentation/
- JSON API – http://jsonapi.org/format/
- Microsoft API Design – https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design
- JIRA REST API – https://developer.atlassian.com/server/jira/platform/jira-rest-api-examples
- Confluece REST API – https://developer.atlassian.com/server/confluence/
- Ebay API – https://developer.ebay.com/api-docs/static/ebay-rest-landing.html
- Oracle REST Data Services – http://www.oracle.com/technetwork/developer-tools/rest-data-services/overview/index.html
- PayPal REST API – https://developer.paypal.com/docs/api/overview/
- Data on the Web Best Practices – https://www.w3.org/TR/dwbp/#intro
- SAP Guidelines for Future REST API Harmonization – https://d.dam.sap.com/m/xAUymP/54014_GB_54014_enUS.pdf
- GitHub API – https://developer.github.com/v3/
- Zalando – https://github.com/zalando/restful-api-guidelines
- Dropbox – https://www.dropbox.com/developers
- Twitter – https://developer.twitter.com/en/docs

Others

- CQRS – https://martinfowler.com/bliki/CQRS.html
- ITU – https://www.itu.int/en/ITU-T/ipr/Pages/open.aspx
- OWASP Rest Security Cheat Sheet – https://www.owasp.org/index.php/REST_Security_Cheat_Sheet
- DDD – https://martinfowler.com/bliki/BoundedContext.html
- REST Principles – https://en.wikipedia.org/wiki/Representational_state_transfer
- Open/Closed Principle – https://en.wikipedia.org/wiki/Open/closed_principle
- Which style of WSDL should I use? – https://www.ibm.com/developerworks/library/ws-whichwsdl/
- https://www.ict.govt.nz/guidance-and-resources/standards-compliance/api-standard-and-guidelines/
- http://www.sabsa.org/node/69
- https://www.owasp.org/index.php/XSS_Prevention_Cheat_Sheet
- https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
- https://www.owasp.org/index.php/Security_by_Design_Principles
- https://www.owasp.org/index.php/OWASP_Top_Ten_Cheat_Sheet
- https://www.owasp.org/index.php/OWASP_API_Security_Project
- https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet
- https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
- https://www.owasp.org/index.php/Query_Parameterization_Cheat_Sheet
- https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf
- http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

[Annexes follow]

ANNEX I - LIST OF WEB SERVICE DESIGN RULES AND CONVENTIONS

The following table summarizes service design rules and conventions, and identifies basic conformance requirements in terms of which conformance level, Web Services API implementation support.

[Note: the table will be completed once all Rules are defined and agreed]

Rule ID	Rule	Conformance Level	Comment
[RS-xx]		A	
…..			
[WS-xx]			
…			

ANNEX II - LIST OF REST API RESOURCES AND QUERY PARAMETERS

The following resource names are recommended for REST Service Contracts. The recommended response data type in XSD format following WIPO Standard ST.96 is also provided below. For the JSON format, the XML to JSON transformation design rules should be followed.

[Note: the table below includes some examples for further discussion and it will be completed with more examples in due course.]

Business Context	Resource Name	Request Data Type	Response Data Type	Possible Query Parameters
Trademark	/trademarks			
Trademark	/images			
Trademark	/applicants			
Trademark	/correspondence-addresses			
Patent	/patents			
Patent	/inventors			
Patent	/assignees			
Patent	/locations			
Patent	/CPC			
Design	/designs			

The following query parameters SHOULD apply to all the REST API services:

Name	Type	Description	Design Rule
format	string	Used for content-type negotiation (prefer a HTTP request header)	[##-##]
v	string	Used for service versioning (prefer indicating version as path segment of the URL)	[##-##]
top	positive integer	The page size used for pagination	[##-##]
offset	positive integer	The offset used for pagination	[##-##]
orderBy	comma-separated list of attributes	Multi-attribute sorting criterion	[##-##]
expand	comma-separated list of attributes	Used for expanding the body of the returned content	[##-##]
count	boolean	Returns the number of elements in a collection (may be inline)	[##-##]
apiKey	string	Used to indicate a Web API Key (a HTTP header should be preferred)	[##-##]

ANNEX III - LIST OF SOAP WEB API NAMES

The following service names are recommended for SOAP Service Contracts. The recommended response data type according to the WIPO Standard ST.96 is also provided.

[Note: the table below includes some examples for further discussion and it will be completed with more examples in due course.]

Service Name	Response Data Type	Description
`PatentsService`	`PatentPublication.xsd`	SOAP web service to manage patents.
`TrademarkApplicationsService`	`TrademarkApplication.xsd`	SOAP web service to manage trademark applications.
`DesignsService`	`Design.xsd`	SOAP web service to manage industrial designs.

ANNEX IV – RESTFUL WEB API MODEL SERVICE CONTRACT

A model service contract following the design rules defined in this standard and based on RAML is provided below. An IP Office will be able to download the RAML and slightly adapt it in order to implement its own API.

- A draft RAML model contract: (draft_annexiv_raml_st96_example)

- *Note: A draft OAS model contract will be developed and added as a separate file in due course.*

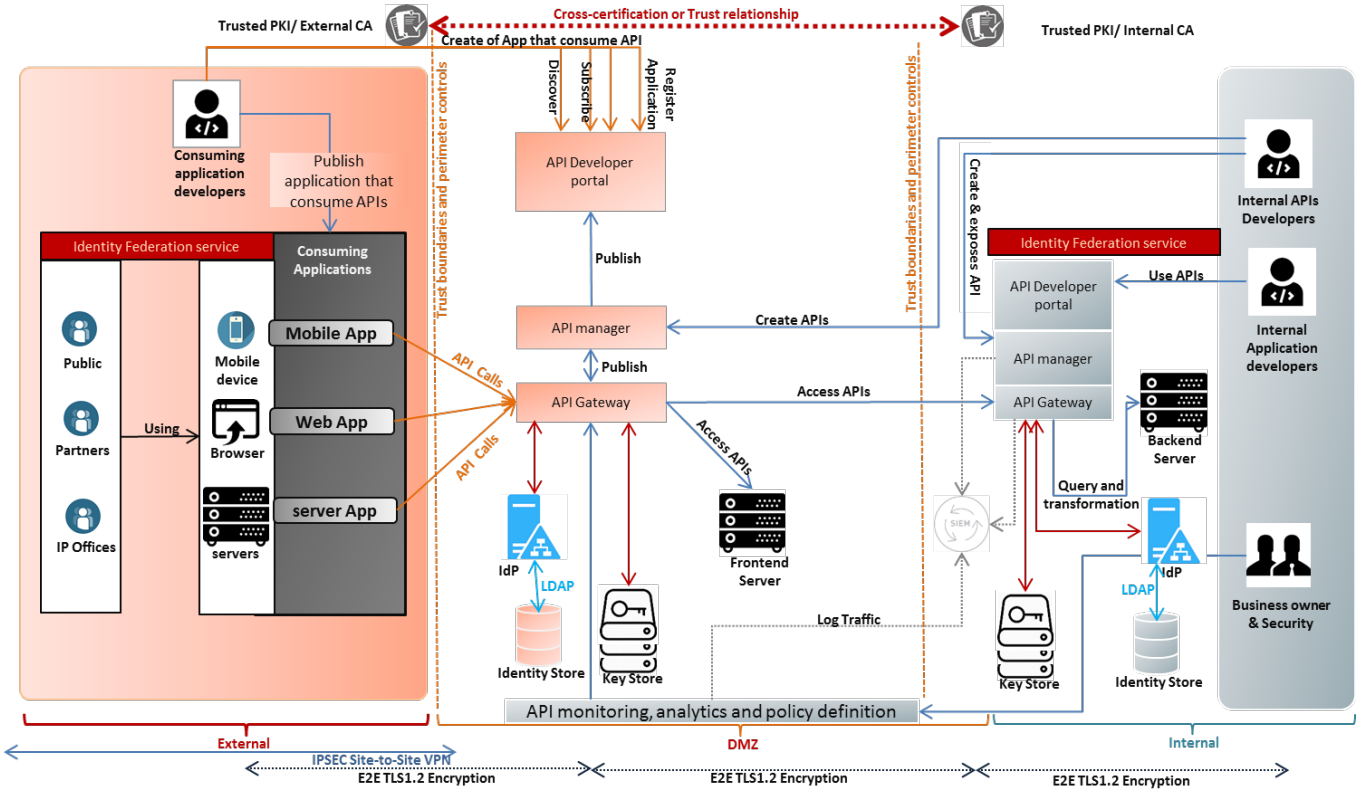
ANNEX V - SOAP WEB API MODEL SERVICE CONTRACT

A model service contract following the design rules defined in this standard and based on WSDL is provided below. An IP Office will be able to download the WSDL and slightly adapt it in order to implement its own API.

- *Note: A draft WSDL model contract will be developed and added as a separate file in due course.*

ANNEX VI – HIGHT LEVEL SECURITY ARCHITECTURE BEST PRACTICES

The security architecture defines the services and mechanisms that should be implemented to enforce defined policies and rules while also providing a framework to further standardize and automate security. The core services and mechanisms of this API Security Framework (the development portal, API manager and API gateway) provide a grouping of functionality. These functions can be delivered by discrete applications, bespoke code development, via COTS products or through leveraging existing technologies that can be configured to provide these functions / services. Some of the functionality may overlap or be combined into one or more products depending on the vendor used.



The recommended security architecture SHOULD have the following API security services and mechanisms:

- A Web API portal to provide functions such as API discovery, API analytics, access to specifications and description including SLAs, social network and FAQs
- A Web API manager to provide centralized API administration and governance for API catalogues, management of registration and on-boarding of various API developer communities, API lifecycle management, application of pre-defined security profiles, and security policies lifecycle management.
- A Web API gateway to provide security automation capabilities including but not limited to centralized threat protections, centralized API authentication, authorization, logging, security policy enforcement, message encryption, monitoring, and analytics.
- A Web API monitoring and analytics service to provide functions such as advanced API services monitoring, analytics, profile usage for security baselines, changes of usage and demand.
- A credential store to provide capabilities to securely store API keys, secrets, certificates, etc.
- A trusted Certificate Authority (CA) to issue secure certificates and enable trust establishment between the various Offices.
- A Security Information and Event Management system (SIEM) to enable security logs correlation and advanced security analytics and monitoring.
- An Identity Provider to manage the identities stored in the LDAP directories and enable authentication.

ANNEX VII – HTTP STATUS CODES

The following response status code categories are defined:

- 1xx: Informational - Communicates transfer protocol-level information
- 2xx: Success - Indicates that the client's request was accepted successfully
- 3xx: Redirection - Indicates that the client must take some additional action in order to complete their request
- 4xx: Client Error - This category of error status codes points the finger at clients
- 5xx: Server Error - The server takes responsibility for these error status codes

The following table consolidates the HTTP Status Codes and provides references to the relative IETF RFCs.

Value	Description	Reference
100	Continue	[RFC7231, Section 6.2.1]
101	Switching Protocols	[RFC7231, Section 6.2.2]
102	Processing	[RFC2518]
103	Early Hints	[RFC8297]
104-199	Unassigned	
200	OK	[RFC7231, Section 6.3.1]
201	Created	[RFC7231, Section 6.3.2]
202	Accepted	[RFC7231, Section 6.3.3]
203	Non-Authoritative Information	[RFC7231, Section 6.3.4]
204	No Content	[RFC7231, Section 6.3.5]
205	Reset Content	[RFC7231, Section 6.3.6]
206	Partial Content	[RFC7233, Section 4.1]
207	Multi-Status	[RFC4918]
208	Already Reported	[RFC5842]
209-225	Unassigned	
226	IM Used	[RFC3229]
227-299	Unassigned	
300	Multiple Choices	[RFC7231, Section 6.4.1]
301	Moved Permanently	[RFC7231, Section 6.4.2]
302	Found	[RFC7231, Section 6.4.3]
303	See Other	[RFC7231, Section 6.4.4]

304	Not Modified	[RFC7232, Section 4.1]
305	Use Proxy	[RFC7231, Section 6.4.5]
306	(Unused)	[RFC7231, Section 6.4.6]
307	Temporary Redirect	[RFC7231, Section 6.4.7]
308	Permanent Redirect	[RFC7538]
309-399	Unassigned	
400	Bad Request	[RFC7231, Section 6.5.1]
401	Unauthorized	[RFC7235, Section 3.1]
402	Payment Required	[RFC7231, Section 6.5.2]
403	Forbidden	[RFC7231, Section 6.5.3]
404	Not Found	[RFC7231, Section 6.5.4]
405	Method Not Allowed	[RFC7231, Section 6.5.5]
406	Not Acceptable	[RFC7231, Section 6.5.6]
407	Proxy Authentication Required	[RFC7235, Section 3.2]
408	Request Timeout	[RFC7231, Section 6.5.7]
409	Conflict	[RFC7231, Section 6.5.8]
410	Gone	[RFC7231, Section 6.5.9]
411	Length Required	[RFC7231, Section 6.5.10]
412	Precondition Failed	[RFC7232, Section 4.2][RFC8144, Section 3.2]
413	Payload Too Large	[RFC7231, Section 6.5.11]
414	URI Too Long	[RFC7231, Section 6.5.12]
415	Unsupported Media Type	[RFC7231, Section 6.5.13][RFC7694, Section 3]
416	Range Not Satisfiable	[RFC7233, Section 4.4]
417	Expectation Failed	[RFC7231, Section 6.5.14]
418-420	Unassigned	
421	Misdirected Request	[RFC7540, Section 9.1.2]
422	Unprocessable Entity	[RFC4918]
423	Locked	[RFC4918]
424	Failed Dependency	[RFC4918]
425	Unassigned	
426	Upgrade Required	[RFC7231, Section 6.5.15]
427	Unassigned	
428	Precondition Required	[RFC6585]
429	Too Many Requests	[RFC6585]
430	Unassigned	
431	Request Header Fields Too Large	[RFC6585]
432-450	Unassigned	
451	Unavailable For Legal Reasons	[RFC7725]
452-499	Unassigned	
500	Internal Server Error	[RFC7231, Section 6.6.1]
501	Not Implemented	[RFC7231, Section 6.6.2]
502	Bad Gateway	[RFC7231, Section 6.6.3]
503	Service Unavailable	[RFC7231, Section 6.6.4]
504	Gateway Timeout	[RFC7231, Section 6.6.5]

505	HTTP Version Not Supported	[RFC7231, Section 6.6.6]
506	Variant Also Negotiates	[RFC2295]
507	Insufficient Storage	[RFC4918]
508	Loop Detected	[RFC5842]
509	Unassigned	
510	Not Extended	[RFC2774]
511	Network Authentication Required	[RFC6585]
512-599	Unassigned	

[End of Annexes and of document]